A Parallel Algorithm for Fixed-length Approximate String-Matching with k-mismatches

Maxime Crochemore[1,2], Costas S. Iliopoulos[1,3] and Solon P. Pissis[1]

[1] Dept. of Computer Science, King's College London, London WC2R 2LS, UK
[2] Institut Gaspard-Monge, Université Paris-Est, 77454 Marne-la-Vallée, France
[3] Digital Ecosystems & Business Intelligence Institute, Curtin University
GPO Box U1987 Perth WA 6845, Australia
{mac,csi,pississo}@dcs.kcl.ac.uk

Abstract. This paper deals with the approximate string-matching problem with Hamming distance. The approximate string-matching with k-mismatches problem is to find all locations at which a query of length m matches a factor of a text of length n with k or fewer mismatches. The approximate string-matching algorithms have both pleasing theoretical features, as well as direct applications, especially in computational biology. We consider a generalisation of this problem, the *fixed-length approximate string-matching with k-mismatches problem*: given a text t, a pattern x and an integer ℓ, search for all the occurrences in t of all factors of x of length ℓ with k or fewer mismatches with a factor of t. We present a practical parallel algorithm of comparable simplicity that requires only $\mathcal{O}(\frac{nm\lceil \ell/w \rceil}{p})$ time, where w is the word size of the machine (e.g. 32 or 64 in practice) and p the number of processors. Thus the algorithm's performance is independent of k and the alphabet size $|\Sigma|$. The proposed parallel algorithm makes use of message-passing parallelism model, and word-level parallelism for efficient approximate string-matching.

Key words: string algorithms, parallel algorithms, approximate string-matching

1 Introduction

The problem of finding factors of a text similar to a given pattern has been intensively studied over the last thirty years and it is a central problem in a wide range of applications, including file comparison, spelling correction, information retrieval, and searching for similarities among biosequences.

One of the most common variants of the approximate string-matching problem is that of finding factors that match the pattern with at most k-differences. The first algorithm addressing exactly this problem is attributable to Sellers [15]. Sellers algorithm requires $\mathcal{O}(mn)$ time, where m is the length of the query and n is the length of the text. One of the first intensive study on the question is by Ukkonen [16]. A thread of practice-oriented results exploited the hardware

word-level parallelism of bit-vector operations. Wu and Manber in [18] showed an $\mathcal{O}(knm/w)$ algorithm for the k-differences problem, where w is the number of bits in a machine word. Baeza-Yates and Navarro in [1] have shown a $\mathcal{O}(knm/w)$ variation on the Wu-Manber algorithm, implying $\mathcal{O}(n)$ performance when $km = \mathcal{O}(w)$. Another general solution based on existing algorithms can be found in [3].

In this paper, we consider the following versions of the sequence comparison problem: given a solution for the comparison of A and $B = b\hat{B}$, can one incrementally compute a solution for A versus $\hat{B}$? and given a solution for the comparison of A and $\hat{B}$, can one incrementally compute a solution for A versus $\hat{B}c$? Here b and c are additional symbols. By solution we mean some encoding of a relevant portion of the traditional dynamic programming matrix D computed for comparing A and B.

Landau, Myers and Schmidt in [8] demonstrated the power of efficient algorithms answering the above questions, with a variety of applications to computational problems such as "the longest common subsequence problem","the longest prefix approximate match problem", "approximate overlaps in the fragment assembly problem", "cyclic string comparison" and "text screen updating".

The above ideas are the bases of the *fixed-length approximate string-matching problem*: given a text t of length n, a pattern x of length m and an integer ℓ, compute the optimal alignment of all factors of x of length ℓ with factors of t. Iliopoulos, Mouchard and Pinzon in [7] presented the Max-Shift algorithm, a bit-vector algorithm that requires $O(nm\lceil \ell/w \rceil)$ time and its performance is independent of k. As such, it can be used to compute blocks of dynamic programming matrix as the 4-Russians algorithm (see [19]).

In this paper, we consider the *fixed-length approximate string-matching with k-mismatches problem*: given a text t, a pattern x and an integer ℓ, search for all the occurrences in t with k or fewer mismatches of all factors of x of length ℓ. There has been ample work in the literature for devising parallel algorithms for different models and platforms, for the approximate string-matching problem [2], [4], [6], [10], [13]. We design and analyse a practical parallel algorithm for addressing the fixed-length approximate string-matching problem with k-mismatches in $\mathcal{O}(\frac{nm\lceil \ell/w \rceil}{p})$ time. Thus the algorithm's performance is independent of k and the alphabet size $|\Sigma|$ (provided that a letter fits in a computer word). The proposed algorithm makes use of message-passing parallelism model, and word-level parallelism for efficient approximate string matching.

The rest of the paper is structured as follows. In Section 2, the basic definitions that are used throughout the paper are presented. In Section 3, we formally define the problem solved in this paper. In Sections 4 and 5, we present the sequential and the parallel algorithm, respectively. In Section 6, we present the experimental results of the proposed algorithm. Finally, we briefly conclude in Section 7.

2 Basic Definitions

A *string* or *sequence* is a succession of zero or more symbols from an alphabet Σ of cardinality s; the string with zero symbols is denoted by ϵ. The set of all strings over the alphabet Σ is denoted by Σ^*. A string x of length m is represented by $x[1\,..\,m]$, where $x[i] \in \Sigma$ for $1 \leq i \leq m$. The length of a string x is denoted by $|x|$. We say that Σ is *bounded* when s is a constant, *unbounded* otherwise. A string w is a factor of x if $x = uwv$ for $u, v \in \Sigma^*$.

Consider the sequences x and y with $x[i], y[i] \in \Sigma \cup \{\epsilon\}$. If $x[i] \neq y[i]$, then we say that $x[i]$ *differs* from $y[i]$. We distinguish among the following three types of differences:

1. A symbol of the first sequence corresponds to a different symbol of the second one, then we say that we have a *mismatch* between the two characters, i.e., $x[i] \neq y[i]$.
2. A symbol of the first sequence corresponds to "no symbol" of the second sequence, that is $x[i] \neq \epsilon$ and $y[i] = \epsilon$. This type of difference is called a *deletion*.
3. A symbol of the second sequence corresponds to "no symbol" of the first sequence, that is $x[i] = \epsilon$ and $y[i] \neq \epsilon$. This type of difference is called an *insertion*.

As an example of the types of differences, see Figure 1.

	1	2	3	4	5	6	7	8
String x:	B	A	D	F	E	ϵ	C	A
	|		|		|		|	|
String y:	B	C	D	ϵ	E	B	C	A

Fig. 1: Types of differences: mismatch in position 2 (A, C), deletion in position 4 (F, ϵ), insertion in position 6 (ϵ, B)

Another way of seeing this difference is that one can transform the x sequence to y by performing operations. The edit distance, $\delta_E(x, y)$, between strings x and y, is the minimum number of operations required to transform x into y. These operations are *Replacement* of a mismatched symbol, a *Deletion* or an *Insertion* of a symbol. The edit distance is symmetrical, and it holds $0 \leq \delta_E(x, y) \leq max(|x|, |y|)$.

Let $t = t[1\,..\,n]$ and $x = x[1\,..\,m]$ with $m \leq n$. We say that x occurs at position q of t with at most k-differences (or equivalently, a *local alignment of x and t at position q with at most k-differences*), if $t[q] \ldots t[r]$, for some $r > q$, can be transformed into x by performing at most k of the following operations: inserting, deleting or replacing a symbol.

The Hamming distance δ_H is defined only for strings of the same length. For two strings x and y, $\delta_H(x, y)$ is the number of places in which the two strings differ, i.e. have different characters. Formally

$$\delta_H(x,y) = \sum_{i=1}^{|x|} 1_{x[i]\neq y[i]} \text{ , where } 1_{x[i]\neq y[i]} = \begin{cases} 1, & \text{if } x[i] \neq y[i] \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

The Hamming distance is symmetrical, and it holds $0 \leq \delta_H(x, y) \leq |x|$.

3 Problem Definition

The focus is on computing matrix M, which contains the number of mismatches of all factors of pattern x of length ℓ and any contiguous factor of the text t of length ℓ.

Example. Let the text $t = x = GGGTCTA$ and $\ell = 3$. Table 1 shows the matrix M.

		0	1	2	3	4	5	6	7
		ϵ	G	G	G	T	C	T	A
0	ϵ	0	0	0	0	0	0	0	0
1	G	1	0	0	0	1	1	1	1
2	G	2	1	0	0	1	2	2	2
3	G	3	2	1	0	1	2	3	3
4	T	3	3	2	1	0	2	2	3
5	C	3	3	3	2	2	0	3	2
6	T	3	3	3	3	2	3	0	3
7	A	3	3	3	3	3	2	3	0

Table 1: Matrix M for $t = x = GGGTCTA$ and $\ell = 3$.

Example. Let the text $t = GTGAACT$, $x = GTCACGT$ and $\ell = 3$. Table 2 shows the matrix M.

The *fixed-length approximate string-matching with at most k-mismatches* problem can be formally defined as follows.

Problem 1. Given a text t of length n, a pattern x of length m and an integer ℓ, find all factors of x of length ℓ that match any contiguous factor of t of length ℓ with at most k-mismatches.

		0	1	2	3	4	5	6	7
		ϵ	G	T	G	A	A	C	T
0	ϵ	0	0	0	0	0	0	0	0
1	G	1	0	1	0	1	1	1	1
2	T	2	2	0	2	1	2	2	1
3	C	3	3	3	1	3	2	2	3
4	A	3	3	3	3	1	2	3	2
5	C	3	3	3	3	3	2	1	3
6	G	3	2	3	2	3	3	2	1
7	T	3	3	1	3	2	3	3	2

Table 2: Matrix M for $t = GTGAACT$, $x = GTCACGT$ and $\ell = 3$.

4 The Bit-Vector-Mismatches Algorithm

Iliopoulos, Mouchard and Pinzon in [7] presented the Max-Shift algorithm, a bit-vector algorithm that solves the *fixed-length approximate string-matching* problem: given a text t of length n, a pattern x of length m and an integer ℓ, compute the optimal alignment of all factors of x of length ℓ and a factor of t. The focus of the Max-Shift algorithm is on computing matrix D', which contains the best scores of the alignments of all factors of pattern x of length ℓ and any contiguous factor of the text t.

The Max-Shift algorithm makes use of word-level parallelism in order to compute matrix D' efficiently, similar to the manner used by Myers in [12]. The algorithm is based on the $\mathcal{O}(1)$ time computation of each $D'[i, j]$ by using bit-vector operations, under the assumption that $\ell \leq w$, where w is the number of bits in a machine word or $\mathcal{O}(\ell/w)$-time for the general case. The algorithm maintains a bit-vector matrix $B[0\,..\,m, 0\,..\,n]$, where the bit integer $B[i, j]$, holds the binary encoding of the path in D' to obtain the optimal alignment at $i,\ j$ with the differences occurring as leftmost as possible.

Here the key idea is to devise a bit-vector algorithm for the *fixed-length approximate string-matching with at most k-mismatches* problem. We maintain the bit-vector $B[i, j] = b_\ell\,..\,b_1$, where $b_\lambda = 1,\ 1 \leq \lambda \leq \ell$, if there is a mismatch of a contiguous factor of the text $t[i - \ell + 1\,..\,i]$ and $x[j - \ell + 1\,..\,j]$ in the λ^{th} position. Otherwise we set $b_\lambda = 0$.

Given the restraint that the integer ℓ is less than the length of the computer word w, then the bit-vector operations allow to update each entry of the matrix B in constant time (using "shift"-type of operation on the bit-vector). The maintenance of the bit-vector is done via operations defined as follows:

1. *shiftc*(x): shifts and truncates the leftmost bit of x.
2. $\delta_H(x, y)$: returns the minimum number of replacements required to transform x into y

The Bit-Vector-Mismatches algorithm for computing the bit-vector matrix B and matrix M is outlined in Figure 2.

Bit-Vector-Mismatches
▷Input: t, n, x, m, ℓ
▷Output: B, M
1 **begin**
2 ▷ Initialisation
3 **for** $i \leftarrow 0$ **until** n **do**
4 $B[0,i] \leftarrow 0$; $M[0,i] \leftarrow 0$
5 **for** $i \leftarrow 0$ **until** m **do**
6 $B[i,0] \leftarrow \min(i,\ell)$ 1's; $M[i,0] \leftarrow \min(i,\ell)$
7 ▷ Matrix B and Matrix M computation
8 **for** $i \leftarrow 1$ **until** m **do**
9 **for** $j \leftarrow 1$ **until** n **do**
10 $B[i,j] \leftarrow \mathit{shiftc}(B[i-1,j-1])$ OR $\delta_H(x[i],t[j])$
11 $M[i,j] \leftarrow \mathit{ones}(B[i,j])$
12 **end**

Fig. 2: The Bit-Vector-Mismatches algorithm for computing matrix B and matrix M

Example. Let the text $t = x = GGGTCTA$ and $\ell = 3$. Table 3 shows the bit-vector matrix B. Consider the case when $i = 7$ and $j = 5$. Cell $B[7,5] = 101$ denotes that factors $t[3\,..\,5] = CTA$ and $t[5\,..\,7] = GTC$ have a mismatch in position 1, a match in position 2, and a mismatch in position 3, resulting in a total of two mismatches, as shown in cell $M[7,5]$ (see Table 1).

		0	1	2	3	4	5	6	7
		ϵ	G	G	G	T	C	T	A
0	ϵ	0	0	0	0	0	0	0	0
1	G	1	0	0	0	1	1	1	1
2	G	11	10	00	00	01	11	11	11
3	G	111	110	100	000	001	011	111	111
4	T	111	111	101	001	000	011	110	111
5	C	111	111	111	011	011	000	111	101
6	T	111	111	111	111	110	111	000	111
7	A	111	111	111	111	111	101	111	000

Table 3: The bit-vector matrix B for $t = x = GGGTCTA$ and $\ell = 3$.

Assume that the bit-vector matrix $B[0\,..\,m, 0\,..\,n]$ is given. We can use the function $ones(v)$, which returns the number of 1's (bits set on) in the bit-vector v, to compute matrix M (see Figure 2, line 11).

Theorem 1. Given the text $t = t[1\,..\,n]$, the pattern $x = x[1\,..\,m]$, the motif length ℓ, and the size w of the computer word, the BIT-VECTOR-MISMATCHES algorithm correctly computes the matrix M in $\mathcal{O}(nm\lceil \ell/w \rceil)$ units of time.

Proof. Without loss of generality, assume that we want to compute cell $M[i, j]$, where

$$M[i, j] = \delta_H(x[i - \ell + 1\,..\,i], t[j - \ell + 1\,..\,j]) \tag{2}$$

It is not difficult to see that,

$$\delta_H(x[i-\ell+1\,..\,i], t[j-\ell+1\,..\,j]) = \delta_H(x[i-\ell+1\,..\,i-1], t[j-\ell+1\,..\,j-1]) + \delta_H(x[i], t[j]) \tag{3}$$

Let $last(b[\ell]\,..\,b[1])$ be an operation that returns the leftmost bit of the bit-vector b. It follows that,

$$\delta_H(x[i-\ell+1\,..\,i-1], t[j-\ell+1\,..\,j-1]) = M[i-1, j-1] - last(B[i-1, j-1]) \tag{4}$$

From Equations 2, 3 and 4,

$$M[i, j] = M[i - 1, j - 1] - last(B[i - 1, j - 1]) + \delta_H(x[i], t[j]) \tag{5}$$

Equation 5 is equivalent to line 10 of the BIT-VECTOR-MISMATCHES algorithm. □

Hence, this algorithm runs in $\mathcal{O}(nm)$ under the assumption that $\ell \leq w$ and its space complexity is reduced to $\mathcal{O}(n)$ by noting that each row of B depends only on its immediately preceding row.

5 The PARALLEL-BIT-VECTOR-MISMATCHES Algorithm

The next proposed parallel algorithm makes use of the message-passing parallelism model by using p processors. The following assumptions for the model of communications in the parallel computer are made. The parallel computer comprises a number of nodes. Each node comprises one or several identical processors interconnected by a switched communication network. The time taken to send a message of size n between any two nodes is independent of the distance between nodes and can be modelled as $t_{comm} = t_s + nt_w$, where t_s is the latency or start-up time of the message, and t_w is the transfer time per data. The links between two nodes are full-duplex and single-ported: a message can be transferred in both directions by the link at the same time, and only one message can be sent and one message can be received at the same time.

We will use the *functional* decomposition, in which the initial focus is on the computation that is to be performed rather than on the data manipulated by

the computation. We assume that both text t and pattern x are stored locally on each processor. This can be done by using a *one-to-all* broadcast operation in $(t_s + t_w(n+m)) \log p$ communication time, which is asymptotically $\mathcal{O}(n \log p)$.

The key idea behind parallelising the Bit-Vector-Mismatches algorithm, is that cell $B[i,j]$ can be computed only in terms of $B[i-1, j-1]$. Based on this, if we partition the problem of computing matrix B (and M) into a set of diagonal vectors $\Delta_0, \Delta_1, .., \Delta_{n+m}$, as shown in Equation 6, the computation of each one of these would be independent, and hence parallelisable.

$$\Delta_\nu[x] = \begin{cases} B[\nu - x,\ x] \ :\ 0 \leq x \leq \nu, & \text{(a)} \\ B[m - x,\ \nu - m + x] \ :\ 0 \leq x < m + 1, & \text{(b)} \\ B[m - x,\ \nu - m + x] \ :\ 0 \leq x < n + m - \nu + 1, & \text{(c)} \end{cases} \tag{6}$$

where,
(a) if $0 \leq \nu < m$
(b) if $m \leq \nu < n$
(c) if $n \leq \nu < n + m + 1$

It is possible that in a certain diagonal Δ_ν, $\nu > 0$, a processor will need a cell or a pair of cells, which were not computed on its local memory in diagonal $\Delta_{\nu-1}$. We need a communication pattern in each diagonal Δ_ν, for all $0 \leq \nu < n + m$, which minimises the data exchange between the processors. It is obvious, that in each diagonal, each processor needs only to communicate with its neighbours. In particular, in each diagonal, each processor needs to swap the boundary cells with its left and right neighbour processor.

An outline of the Parallel-Bit-Vector-Mismatches algorithm in each diagonal Δ_ν, for all $0 \leq \nu < n + m + 1$, is as follows:

Step 1. Each processor is assigned with $|\Delta_\nu|/p$ cells (without loss of generality).
Step 2. Each processor computes each allocated cell using the Bit-Vector-Mismatches algorithm.
Step 3. Processors communication involving point-to-point boundary cells swaps.

Theorem 2. Given the text $t = t[1..n]$, the pattern $x = x[1..m]$, the motif length ℓ, the size w of the computer word, and the number of processors p, the Parallel-Bit-Vector-Mismatches algorithm computes the matrix M in $\mathcal{O}(\frac{nm\lceil \ell/w \rceil}{p})$ units of time.

Proof. We partition the problem of computing matrix B into a set of $n + m + 1$ diagonal vectors, thus $\mathcal{O}(n)$ supersteps. In step 1, the allocation procedure runs in $\mathcal{O}(1)$ time. In step 2, the cells computation requires $\mathcal{O}(\frac{m\lceil \ell/w \rceil}{p})$ time. In step 3, the data exchange between the processors involves $\mathcal{O}(1)$ point-to-point message transfers. Hence, asymptotically, the overall time is $\mathcal{O}(\frac{nm\lceil \ell/w \rceil}{p})$. □

Hence, the parallel algorithm runs in $\mathcal{O}(\frac{nm}{p})$ under the assumption that $\ell \leq w$, and its space complexity is reduced to $\mathcal{O}(n)$ by noting that each diagonal vector Δ_ν of matrix B, for all $2 \leq \nu \leq n + m$, depends only on $\Delta_{\nu-2}$.

6 Experimental Results

In order to evaluate the parallel efficiency of our algorithm, we implemented the Bit-Vector-Mismatches algorithm in ANSI *C* language and parallelised it with the use of the *MPI* library. Both implementations, the sequential and the parallel algorithm, are available at a website[1], which has been set up for maintaining the source code and the documentation.

Experimental tests were run on 1 up to 16 processing nodes (2.6 GHz AMD Opteron) of a cluster architecture. As an input, DNA sequences of the mouse chromosome X were used, retrieved from the *Ensembl* genome database. Experimental results regarding the execution time and measured speed-up are illustrated in Figures 3 and 4, respectively. The speed-up is calculated as the ratio of elapsed time with p processors to elapsed time with one processor.

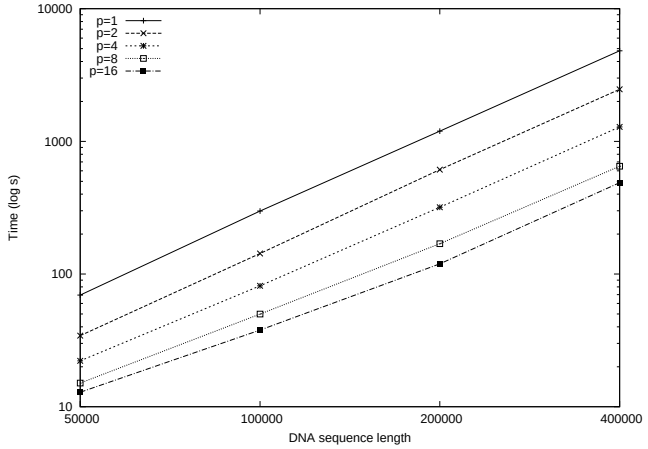

Fig. 3: Execution time for $t = x$ and $\ell = 20$

The presented experimental results demonstrate a good scaling of the code. The proposed algorithm scales well even for small problem sizes. As expected in some cases, when increasing the problem size, the algorithm achieves a linear speed-up, confirming our theoretical results. Further tests were conducted for different values of fixed-length ℓ, with no difference observed, regarding the execution time.

7 Conclusion

We have presented a practical parallel algorithm that solves a generalisation of the approximate string-matching problem. In particular, the proposed parallel algorithm solves the fixed-length approximate string matching with k-mismatches

[1] http://www.dcs.kcl.ac.uk/pg/pississo/

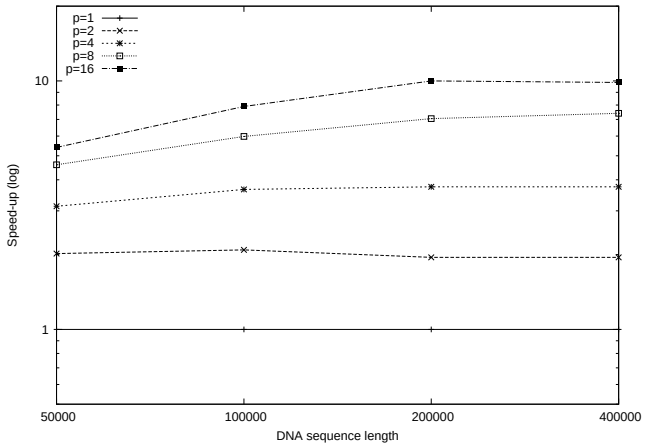

Fig. 4: Measured speed-up for $t = x$ and $\ell = 20$

problem in $\mathcal{O}(\frac{nm\lceil \ell/w \rceil}{p})$ time, which is $\mathcal{O}(\frac{nm}{p})$, in practical terms. It is considerably simple and elegant, it achieves a theoretical and practical linear speed-up, it does not require text preprocessing, it does not use/store look up tables and it does not depend on the number of differences k and the alphabet size $|\Sigma|$.

References

1. Baeza-Yates, R. A., Navarro, G.: A faster algorithm for approximate string matching. In *CPM*, volume 1075 of Lecture Notes in Computer Science, pages 1–23. Springer-Verlag (1996)
2. Bertossi, A. A., Luccio, F., Pagli, L., Lodi, E.: A parallel solution to the approximate string-matching problem. *The Computer Journal*, 35(5):524–526. (1992)
3. Crochemore, M., Iliopoulos, C. S., Pinzon, Y. J: Speeding up Hirschberg and Hunt-Szymanski LCS algorithms. *Fundamenta Informaticae*, 56(1,2):89–103. (2002)
4. Galper, A. R., Brutlag, D. R.: Parallel similarity search and alignment with the dynamic programming method. *Technical Report KSL 90–74*. Stanford University, 14pp. (1990)
5. Hall, N.: Advanced sequencing technologies and their wider impact in microbiology. *J. Exp. Biol.*, 210(pt 9):1518–1525. (2007)
6. Huang, X.: A space-efficient parallel sequence comparison algorithm for a Message-Passing Multiprocessor. *International Journal of Parallel Programming*, 18(3):223–239. (1990)
7. Iliopoulos, C. S., Mouchard, L., Pinzon, Y. J.: The Max-Shift algorithm for approximate string matching. In *WAE '01: Proceedings of the 5th International Workshop on Algorithm Engineering*, pages 13–25. (2001)
8. Landau, G., Myers, G., Schmidt, J.P., Schmidt, P.: Incremental String Comparison. *SIAM Journal on Computing*, 27:557–582. (1995)
9. Landau, G. M., Vishkin, U.: Fast string matching with k differences. *Journal of Computer and Systems Sciences*, 37(1):63–78. (1988)

10. Landau, G. M., Vishkin, U.: Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169. (1989)
11. Margulies, E. H., Birney, E.: Approaches to comparative sequence analysis: towards a functional view of vertebrate genomes. *Nat. Rev. Genet.*, 9(4):303–313. (2008)
12. Myers, E. W.: A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming. *Journal of the ACM*, 46:395–415. (1999)
13. dos Reis, C.C.T., Approximate string-matching algorithm using parallel methods for molecular sequence comparisons. In *Artificial intelligence, 2005. epia 2005. portuguese conference on*, pages 140–143. (2005)
14. Schuster, S. C.: Next-generation sequencing transforms today's biology. *Nature Methods*, 5(1):16–18. (2007)
15. Seller, P. H.: The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms*, 1(4):359–373. (1980)
16. Ukkonen, E.: Finding approximate patterns in strings. *J. of Algorithms*, 6(1):132–137. (1985)
17. Wold, B., Myers, R.: Sequence consensus methods for functional genomics. *Nature Methods*, 5(1):19–21. (2007)
18. Wu, S., Manber, U.: Fast text searching allowing errors. *CACM*, 35(10):83–91. (1992)
19. Wu, S., Manber, U., Myers, G.: A subquadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67. (1996)